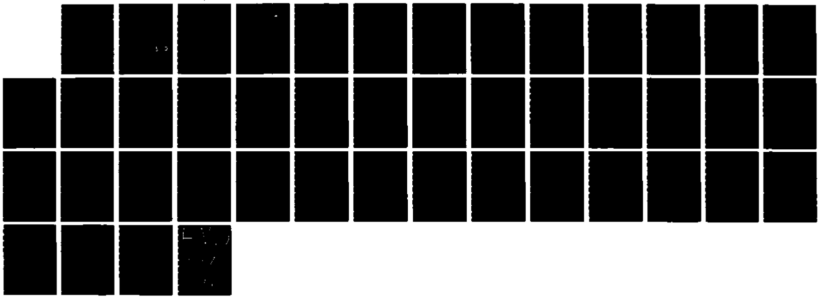
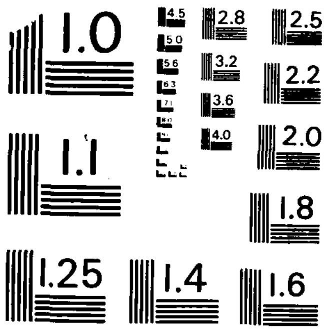
AD-A175 253   MICROCODE VERIFICATION USING SDVS (STATE DELTA VERIFICATION SYSTEM): THE..(U) AEROSPACE CORP EL SEGUNDO CA LAB OPERATIONS B H LEVY 30 AUG 84   1/1

UNCLASSIFIED   TR-0084(4778)-1 SD-TR-86-54   F/G 9/2   NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS - 1963 - A

(12)

REPORT SD-TR-86-54

Microcode Verification Using SDVS: The Method and a Case Study

B. H. LEVY
Information Sciences Research Office
Laboratory Operations
The Aerospace Corporation
El Segundo, CA 90245

1 October 1986

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

Prepared for

NATIONAL SECURITY AGENCY
9800 Savage Rd.
Fort George C. Meade, MD 20755

SPACE DIVISION
AIR FORCE SYSTEMS COMMAND
Los Angeles Air Force Station
P.O. Box 92960, Worldway Postal Center
Los Angeles, CA 90009-2960

86 12 [illegible]

THE AEROSPACE CORPORATION

DOCUMENT CHANGE NOTICE

TO	Distribution	DATE	9 December 1986
SUBJECT	TR-0086A(2778)-1	FROM	Publications

Please change the report number of this document to TR-0084(4778)-1 (block 6, DD Form 1473). This report and the work described therein were completed 30 August 1984.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER SD-TR-86-54	2. GOVT ACCESSION NO. ADA175252	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* MICROCODE VERIFICATION USING SDVS: THE METHOD AND A CASE STUDY		5. TYPE OF REPORT & PERIOD COVERED 6. PERF[illegible] UMBER [illegible] TR-00[illegible]-1
7. AUTHOR(s) B. H. Levy		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS The Aerospace Corporation El Segundo, Calif. 90245		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Space Division Los Angeles Air Force Station Los Angeles, Calif. 90009-2960		12. REPORT DATE 1 October 1986 30 Aug [illegible] 13. NUMBER OF PAGES 40
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*		15. SECURITY CLASS. *(of this report)* Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

verification
microcode
symbolic execution
ISPS
correctness proofs
theorem provers
formal machine descriptions

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes SDVS (State Delta Verification System), its application to microcode verification, and the verification of a particular example referred to as the H-machine example. The example illustrates how particular microcode that interprets a computer instruction set can be proved correct and how this proof is accomplished with an existing, automated verification system.

DD FORM 1473 (FACSIMILE)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

CONTENTS

FIGURES

PREFACE

This paper describes SDVS (State Delta Verification System), its application to microcode verification. and the verification of a particular example referred to as the H-machine example The example illustrates how particular microcode that interprets a computer instruction set can be proved correct and how this proof is accomplished with an existing, automated verification system. A shortened version of this paper will appear in the Proceedings of the 17th Annual Microprogramming Workshop, IEEE Computer Society, New Orleans, October 30-November 2, 1984.

I would like to acknowledge the following people at The Aerospace Corporation who developed the state delta theory, implemented SDVS, and were exceedingly responsive to my questions and requests: Eve Cohen, Jeff Cook, Steve Crocker, Jaisook Landauer, Leo Marcus, Hilarie Orman, and Bill Overman (no longer at The Aerospace Corp.). I would also like to thank Mike Vahey and Ben Cohen from Hughes Aircraft Co. for suggesting this example and for their encouragement.

SECTION I
BACKGROUND

The description of SDVS given in this report is not intended to be complete and is provided as background material. Other documents describe SDVS in greater detail[2154]. The report was written with the assumption that the reader is not familiar with correctness proofs, but is familiar with mathematical logic.

A. A NEED FOR NEW VERIFICATION METHODS

Computer systems are becoming too complex to verify using current methods. Hardware and software sophistication have been increasing more rapidly than the sophistication of verification methodologies, creating an ever widening gap between technology and verification capability. As a result, we cannot be confident that the systems being developed properly implement system specifications. Verification problems are costly, causing unnecessary expenditures of time, labor, and hardware.

Systems are currently verified using testing and simulation, methods which are not adequate for VLSI/VHSIC technology and today's large software projects. With testing or simulation a system is deemed correct if correct results are produced for some small sample of input. The complexity of today's systems makes thorough testing impossible because of the large number of potential inputs and execution paths. Thus, systems are delivered and put into operation with a strong likelihood of undetected errors.

Excessive costs and risks will continue to be incurred unless new verification methods are developed. As an alternative to testing our goal is to prove mathematically that an implementation satisfies its specification (or requirements) for *any* input. Both the theory and the tools are being developed to do these correctness proofs for computer system implementations. The verification system being developed at The Aerospace Corp. is called SDVS (State Delta Verification System).

B. CORRECTNESS CRITERION

Testing does not require formal specifications of system requirements. This permits requirements and design documents to be written in English, which can be imprecise and ambiguous. Further, it is generally impractical or impossible to test all potential input. Consequently, with testing there is no systematic and consistent way to show that the implementation of a system satisfies the system requirements.

To eliminate ambiguity and lack of precision, formal languages are used to describe system requirements. Formal languages are a prerequisite for formal mathematical proofs. The correctness criterion is a mathematical theorem stated in a formal language. The theorem states that the properties of the system specification are preserved by the implementation (i.e.. the specification is consistent with the implementation).

C. CORRECTNESS PROOFS

The correctness proof constitutes a verification of the implementation and is a proof of the correctness criterion. This proof is usually done with computer assistance because of the detail involved. Mechanical theorem provers range in performance from trying to discover a proof to checking a proof created manually.

A formal logical language is required for the proof. Programming languages can be used as formal languages for both system specifications and implementations. For example, the behavior of a computer system is typically specified in a programming language (e.g., ISPS[1]). If this programming language can be executed (or translated into an executable language) the behavior of a computer described this way can be simulated or a software implementation written in the programming language can be tested. If the properties of the programming language are formally defined (axiomatized), theorems might be proved about an implementation or a specification written in the programming language. SDVS is used to perform these proofs.

Proofs in SDVS involve symbolic execution, expression simplification, and induction. These topics are discussed in the following sections. Sections on program states, rules for symbolic execution, and state delta specification of ISPS explain the formalism necessary for symbolic execution in SDVS.

We have also incorporated a mapping construct into SDVS. Among other things, this allows us to describe how a computer architecture is implemented by microcode operating on some hardware. A brief description of mapping is given in the section following the discussion on induction.

1. Symbolic Execution

Symbolic execution combined with expression simplification (theorem provers) permits a greater range of input evaluation than can be done practically with testing*. Whereas in testing a data value must be supplied for each input variable, input variables are not set to values in symbolic execution.**

The following example demonstrates the difference between testing and symbolic execution. The programming language used in the example and in SDVS is ISPS. ISPS is a high level programming language

* Symbolic execution is sometimes called the dynamic part of the proof and expression simplification is called the static part of the proof

** Another way of viewing symbolic execution is that symbols are used to represent program inputs and a machine performs symbolic operations on symbols.

commonly used to describe a computer's architecture. The following ISPS program swaps the contents of two variables declared as registers (bitstrings):

```
REGSWITCH (X<15:0>, Y<15:0>) :=
     BEGIN

          ** DECLARATION SECTION **
          TEMP<15:0>

          ** SWAPPING FUNCTION **
          ENTRY {MAIN} :=
               BEGIN
                    TEMP = X    NEXT
                    X = Y     NEXT
                    Y = TEMP
               END
          END
```

To test or simulate this program. data values are supplied for variables X and Y. For example, if X and Y are initialized to values 4 and 1, respectively, then after the execution of REGSWITCH the values of X and Y are 1 and 4, respectively.

On the other hand, there is no initialization in symbolic execution. After symbolic execution, the new value of X is set to the old value of Y and the new value of Y is set to the old value of X. This is written as (#X=.Y & #Y=.X) where a period placed before a variable indicates the value of the variable before execution and a pound sign ("#") before a variable indicates the value of the variable after execution. Refer to the formula (#X=.Y & #Y=.X) as Formula 1 in subsequent discussions.

Formula 1 specifies the behavior of the ISPS procedure REGSWITCH. Such formulas are written in a first-order language with equality. The logical symbols of the language are & (and), $\vee$ (or), $\Rightarrow$ (implies), $\neg$ (not), = (equal), and **if-then-else**

The nonlogical symbols with a fixed interpretation are taken from four quantifier-free theories: integers, bitstrings, arrays, and coverings (set partitions). These theories define the data types in ISPS. The non-logical symbols and their signatures (types) are:

1) **integers**

predicate symbols	signature	description
$\leq$	integer $\times$ integer $\rightarrow$ boolean	less than or equal
$\geq$	integer $\times$ integer $\rightarrow$ boolean	greater than or equal
$<$	integer $\times$ integer $\rightarrow$ boolean	less than
$>$	integer $\times$ integer $\rightarrow$ boolean	greater than

function symbols	signature	description

+	integer X integer → integer	addition
-	integer X integer → integer	subtraction
-	integer → integer	arithmetic negation
*	integer X integer → integer	multiplication
max	integer X integer → integer	maximum
min	integer X integer → integer	minimum
↑	integer X integer → integer	exponentiation
/	integer X integer → integer	division
\	integer X integer → integer	remainder
ICONS	integer X integer X integer → integer	integer construction

constant symbols	signature	description
...-2 -1 0 1 2...	integer	the integers

2) **bitstrings**

function symbols	signature	description
lh	bitstring → non-negative integer	length
usval	bitstring → non-negative integer	value
ussub	bitstring X integer X integer → bitstring	substring
usconc	bitstring X bitstring → bitstring	concatenation
useql	bitstring X bitstring → bitstring	equality
usneq	bitstring X bitstring → bitstring	non-equality
uslss	bitstring X bitstring → bitstring	less than
usleq	bitstring X bitstring → bitstring	less than or equal
usgtr	bitstring X bitstring → bitstring	greater than
usgeq	bitstring X bitstring → bitstring	greater than or equal
usplus	bitstring X bitstring → bitstring	addition
usdifference	bitstring X bitstring → bitstring	subtraction
usnot	bitstring X bitstring → bitstring	logical negation
ustimes	bitstring X bitstring → bitstring	multiplication
usquotient	bitstring X bitstring → bitstring	quotient
usremainder	bitstring X bitstring → bitstring	remainder
usnot	bitstring → bitstring	logical negation
usand	bitstring X bitstring → bitstring	logical conjunction
usor	bitstring X bitstring → bitstring	logical disjunction
usxor	bitstring X bitstring → bitstring	logical exclusive disjunction
useqv	bitstring X bitstring → bitstring	negation of ***usxor***
zeros	integer → bitstring	bitstring of all zeros
ones	integer → bitstring	bitstring of all ones
lastone	bitstring → bitstring	bitstring low-order 1 index

constant symbols	signature	description
bs(x, y) where X and y are integer constants	bitstring	constant bitstring of value X and length Y

3) arrays

function symbols	signature	description
slice	array $\times$ integer $\times$ integer $\rightarrow$ array	select subarray
element	array $\times$ integer $\rightarrow$ U	select array element
where U $\in$ {integers, bitstrings, arrays, coverings}		
range	array $\rightarrow$ integer	array length
aconc	array $\times$ array $\rightarrow$ array	array concatentation

4) coverings (set partitions)

predicate symbols	signature	description
covering	$\text{set}^+ \rightarrow$ boolean	set partition
pcovering	$\text{set}^+ \rightarrow$ boolean	partial set partition
alldisjoint	$\text{set}^+ \rightarrow$ boolean	pairwise disjoint

constant symbols	signature	description
emptyset	set	the empty set

The axioms for the four theories are not listed in this report, but can be found in the SDVS reference manual. Theorems about ISPS programs are deduced from the axioms and the rules for symbolic execution which are explained below.

a. Program States

Formula 1 is called a *postcondition*. One says that the result of symbolically executing REGSWITCH is given by the postcondition or that the postcondition is consistent with the program REGSWITCH. The postcondition is a component of a state of the ISPS program. The *state* of the program is defined as:

1) the current value of the program's variables (including the postcondition)

2) the value of the program counter

3) the path condition (the set of assumptions previously made)

A boolean expression is added to the path condition whenever a decision is encountered in the program. Formula 1 describes the current value of the program's variables when the program counter is 12 (the last instruction) and the path condition is the formula **true** (i.e., there are no branches in the program). There is a state change after the execution of each ISPS instruction, and thus, the program counter is a component of a state. Formula 1 specifies values of variables in the final state.

The state changes are specified by defining a rule for symbolic execution for each program statement. These rules must be sound and agree with the programming language semantics.* For example, in the

* These rules are often used as the definition of the semantics.

procedure REGSWITCH there are three assignment statements, whose execution results in three state changes. The rule for symbolic execution of the assignment statement is used below to demonstrate symbolic execution.

b. Comparison of Rules for Symbolic Execution with Rules for Backward Substitution

Most verification systems are based on the "Hoare proof system."[3] The Hoare proof system is briefly described below in terms of the assignment statement and compared to SDVS, SDVS being a variation of the Hoare proof system.

In the Hoare proof system there is a formula of the form P{S}Q for each programming language statement S. The formula P{S}Q asserts that if formula P is true before executing S and S halts, then the formula Q is true after executing S. P is called the precondition and Q is called the postcondition. These formulas comprise the set of axioms describing the programming language behavior and are used to generate lemmas. The proof of the lemmas guarantee a program is consistent with its specification.

For example, the execution of the assignment statement X=e, where X is a program variable and e is an expression, results in a state change in which the value of variable X is changed, the program counter is incremented, and the path condition remains unchanged. In more detail, all occurrences of variables in expression e are replaced with their values and the new value of variable X is the evaluated expression e. The backward substitution axiom for assignment is $P^X_e\{X=e\}P$, where P^X_e means substitute the expression e for all free occurrences of X in formula P.

A specification for the correctness criteria for REGSWITCH written in the Hoare proof system is (X=.X & Y=.Y {TEMP=X NEXT X=Y NEXT Y=TEMP} X=.Y & Y=.X). With one additional rule we can prove that this formula is true. The additional rule, commonly referred to as the composition rule, specifies the behavior of a sequence of instructions. The rule is written

$$\frac{P\{Q\}R,\ R\{S\}T}{P\{Q \text{ NEXT } S\}T}$$

and means that if the formulas above the line are true (all axioms are true), then the formula below the line is true.

A Hoare style proof of the correctness of REGSWITCH is as follows:

1) X=.Y & TEMP=.X {Y=TEMP} X=.Y & Y=.X (assignment axiom)

2) Y=.Y & TEMP=.X {X=Y} X=.Y & TEMP=.X (assignment axiom)

3) Y=.Y & TEMP=.X {X=Y NEXT Y=TEMP} X=.Y & Y=.X
(1,2, and composition rule)

4) Y=.Y & X=.X {TEMP=X} Y=.Y & TEMP=.X (assignment axiom)

5) Y=.Y & X=.X {TEMP=X NEXT X=Y NEXT Y=TEMP} X=.Y & Y=.X
(3,4, and composition rule)

This proof is typically displayed as an annotated flowchart as in Figure 1.

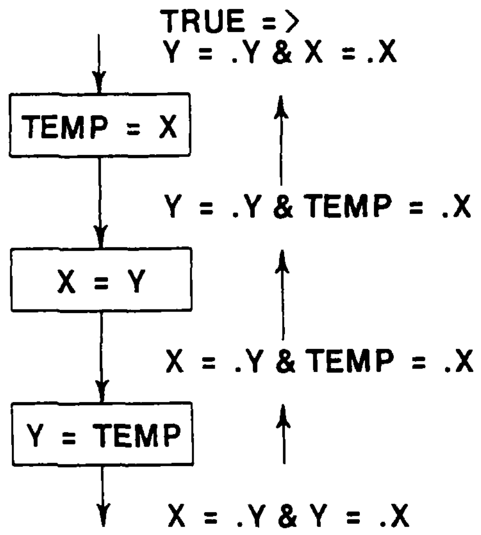


Figure 1: Hoare style proof of REGSWITCH

In summary, the Hoare proof system consists of axioms for each programming language statement (e.g., assignment axiom) and rules for inferring new theorems from the axioms (e.g., composition rule). In Figure 1 we started with the postcondition pushing it backward through the program resulting in the weakest precondition, i.e., the minimum assumption necessary for the postcondition. Alternatively, we could have started with the precondition pushing it through in the forward direction resulting in the strongest postcondition, a total specification of the variables' values in the final state. This is called *symbolic execution*. Figure 2 shows the proof of REGSWITCH using symbolic execution.

The major difference between backward substitution and symbolic execution using SDVS is that partial correctness is usually proved using backward substitution and total correctness is proved with SDVS. *Partial correctness* means that a program is proved correct assuming that the program halts. With SDVS's symbolic execution only halting programs are proved correct. For non-halting programs the symbolic execution does not halt and the postcondition is never satisfied.

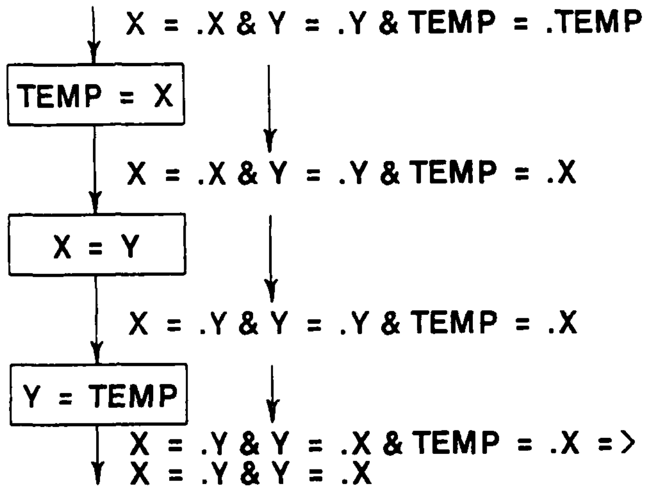

X = .X & Y = .Y & TEMP = .TEMP
TEMP = X
X = .X & Y = .Y & TEMP = .X
X = Y
X = .Y & Y = .Y & TEMP = .X
Y = TEMP
X = .Y & Y = .X & TEMP = .X =>
X = .Y & Y = .X

Figure 2: Symbolic Execution of REGSWITCH

c. State Delta Specification of ISPS

In SDVS, ISPS programs are symbolically executed. Axioms describing the symbolic execution of each ISPS instruction are formulas called *state deltas*. A state delta is of the form

[pre: P
env: E
mod: M
post: Q]

where P is the precondition and Q is the postcondition, both written in the first-order language described above. P and Q are analogous to the precondition and postcondition in formulas of the Hoare proof system. As each formula P{S}Q defines a state change, each state delta also specifies a state change. The programming language statement, S, is not explicitly stated in the state delta. This means the state delta is a formula in which no distinction is made between control and data. "E" and "M" in the state delta specification are lists of program variables and/or program counters. E and M are not necessary for symbolic execution, but facilitate the application of state deltas for symbolic execution.

A state delta specification is defined for each ISPS statement. These are axioms of the proof system. Symbolic execution is the application of these axioms in a proof. A state delta can be used in a proof if the variables in its environment list E have remained unchanged since the state delta was first proved, and if the precondition P is true in the current state. The result of the state delta application is that the postcondition Q will be true at some later time at which the values of no variables other than those listed in the modification list M have changed.

The application of a state delta in a proof can be explained in terms of the timeline shown in Figure 3 where time t1 comes before t2 and t2 before t3.

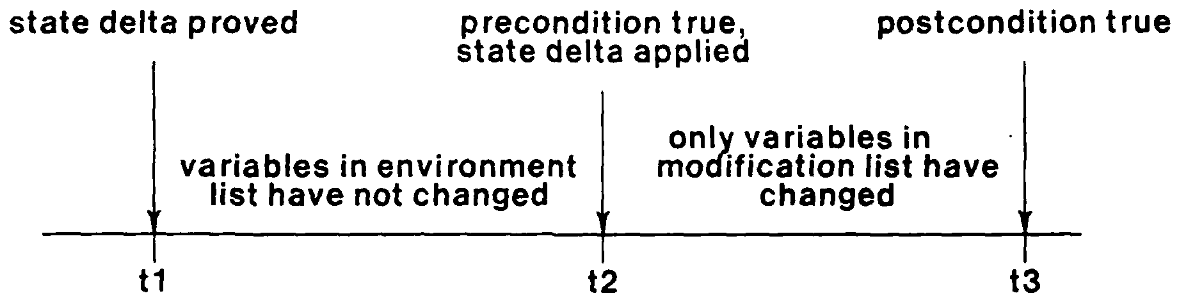

state delta proved
precondition true, state delta applied
postcondition true
variables in environment list have not changed
only variables in modification list have changed
t1
t2
t3

Figure 3: State Delta Application

A formula expressed as a state delta is true at t1. This formula can be used in a proof at a later time t2 if the precondition becomes true and the variables in the environment list have not changed between t1 and t2. The application of the state delta results in a state change at time t3 where the new state is consistent with the postcondition and only the variables in the modification list have changed. The modification list relieves the pre- and postcondition of the burden of copying static information (information that does not change during the state change) forward through the proof. In the context of microcode verification, the environment list may relieve the pre- and postcondition of the need to contain control information (e.g., the ISPS program counter) and also the context in which the state delta was first proved (e.g., ROM cannot change before the state delta is applied in a proof).

Examine again the body of the REGSWITCH procedure. This computation can be represented by three state deltas where the ISPS program counter (upc) is explicit, or by one state delta where upc is implicit from the state delta's nested structure. Figure 4 depicts the three assignment statements as three state deltas and Figure 5 depicts the same program segment as one state delta.

As shown in Figure 5, a state delta can be part of the postcondition of another state delta. When this occurs, it means that if the precondition of the main state delta is true, the state delta(s) in the postcondition of the state delta is(are) also true and may be applied. The state delta in the postcondition in Figure 5 specifies the next instruction, replacing the explicit reference to upc. Because upc is in both the environment and modification lists of each state delta in Figure 5, each of these state deltas can only be applied once in a proof. If the three assignment statements were embedded in a repeat loop their associated state deltas would be applied repeatedly in a proof; upc would not be listed in any of the state deltas' environment lists.

The nested state delta structure is useful if the translation of an ISPS program to state deltas is incremental; i.e., just before an ISPS instruction is symbolically executed, its state delta is generated. This is

```
[pre: .REGSWITCH\upc = 8                                   State Delta 1
env:
mod: REGSWITCH\upc, TEMP
post: #TEMP = .X & #REGSWITCH\upc =9]

[pre: .REGSWITCH\upc = 9                                   State Delta 2
env:
mod: REGSWITCH\upc, X
post: #X = .Y & #REGSWITCH\upc = 10]

[pre: .REGSWITCH\upc = 10                                  State Delta 3
env:
mod: REGSWITCH\upc, Y
post: #Y = .TEMP & #REGSWITCH\upc = 11]
```

Figure 4: REGSWITCH Expressed as Three State Deltas with Explicit Program Counter

```
[pre: REGSWITCH\upc = 8                                    SD 1
env: REGSWITCH\upc
mod: REGSWITCH\upc, TEMP
post: #TEMP = .X & [pre: true                              SD 1.1
                    env: REGSWITCH\upc
                    mod: REGSWITCH\upc, X
                    post: #X = .Y & [pre: true             SD 1.2
                                     env: REGSWITCH\upc
                                     mod: REGSWITCH\upc, Y
                                     post: #Y = .TEMP]]]
```

Figure 5: REGSWITCH Expressed as One State Delta with Implicit Program Counter

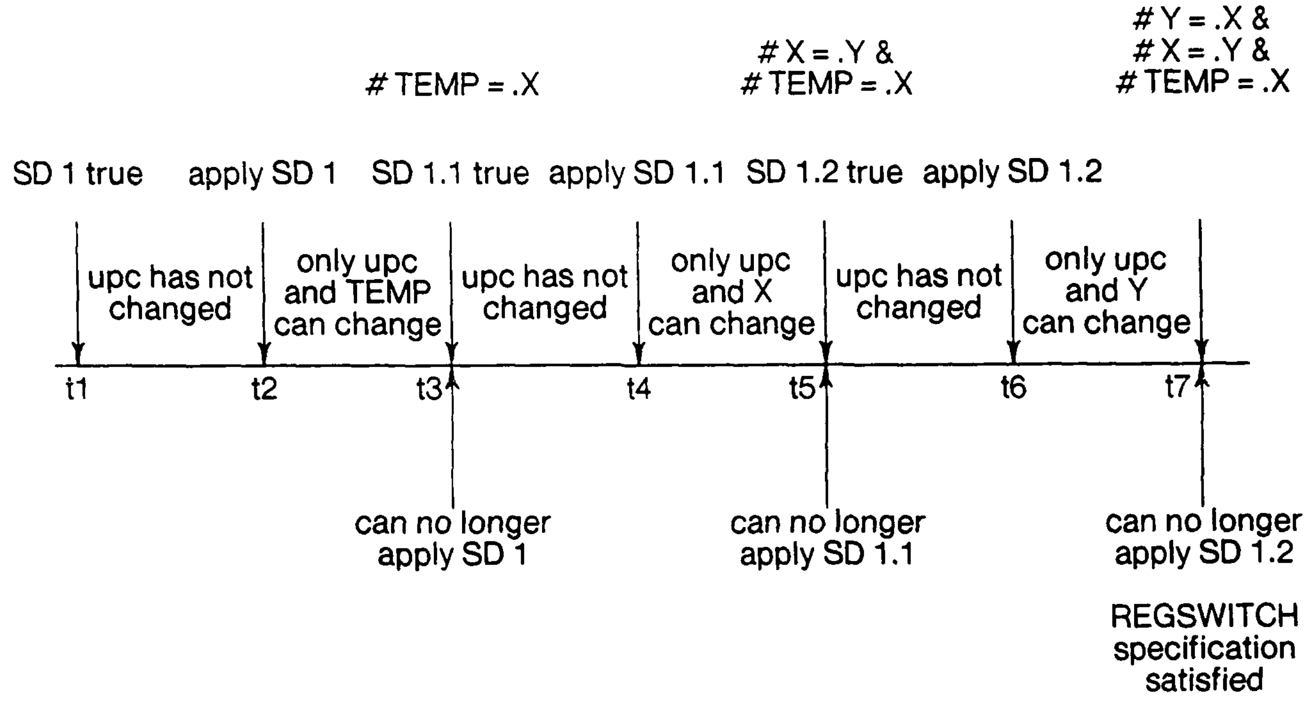
desirable for reducing execution time and storage (SDVS has an incremental ISPS translator). If $S_1 ... S_n$ are ISPS statements and **TR** is a function that translates ISPS programs to state deltas then the translation of a sequential program segment is **TR**(S_1 NEXT...NEXT S_n)= **TR**(S_1)**TR**(S_2 NEXT...NEXT S_n). Figure 6 shows the first incremental translation of the three assignment statements in REGSWITCH.

```
[pre: REGSWITCH\upc = 8
env: REGSWITCH\upc
mod: REGSWITCH\upc, TEMP
post: #TEMP = .X & TR(X = Y NEXT Y = TEMP)]
```

Figure 6: Incremental Translation of REGSWITCH

We have demonstrated the translation of assignment statements and sequential program segments to state deltas. Similarly, there are translations for branching, loops and procedure calls.

The proof of REGSWITCH using state deltas (in the nested structure) as axioms of the proof system is



#TEMP = .X
#X = .Y & #TEMP = .X
#Y = .X & #X = .Y & #TEMP = .X
SD 1 true
apply SD 1
SD 1.1 true
apply SD 1.1
SD 1.2 true
apply SD 1.2
upc has not changed
only upc and TEMP can change
upc has not changed
only upc and X can change
upc has not changed
only upc and Y can change
t1
t2
t3
t4
t5
t6
t7
can no longer apply SD 1
can no longer apply SD 1.1
can no longer apply SD 1.2
REGSWITCH specification satisfied

Figure 7: SDVS Proof of REGSWITCH

shown in Figure 7. The state deltas specifying the three assignment statements are used in the proof at times t2, t4, and t6. Because upc is in the modification list, upc may change value. Once axiom SD1, SD 1.1, or SD 1.2 is used it cannot be used again because upc is also in the environment list.

2. Symbolic Expression Simplification

Notice that the result of symbolically executing REGSWITCH is the strongest postcondition (#Y = .X & #X = .Y & #TEMP = .X), and the formula specifying the behavior of REGSWITCH is (#Y=.X & #X=.Y). Therefore, the following lemma must be proved to show the specification is consistent with the program: ((#Y=.X & #X=.Y & #TEMP=.X) $\Rightarrow$(#Y=.X & #X=.Y)). This is trivial to show, but for more complex programs the lemmas generated from either a Hoare proof system or SDVS may require lengthy proofs using the axioms in the first-order language. Symbolic execution (the application of symbolic execution axioms written as state deltas) is completely automated. However, the simplification of a symbolic expression and the proof of consistency with the postcondition is partially automated using a theorem prover. Simplification is an interactive process and may require user assistance. The theorem prover uses axioms in the theories of integers, bitstrings, arrays, and coverings. SDVS automatically applies some of the axioms in proofs via pattern matching and demons. Other axioms are invoked by the user when needed, thus avoiding unnecessary expression simplification and reducing execution time.

3. Induction

In order to prove theorems about programs containing loops, we use induction over a set ordered like natural numbers; we want to show without testing each possibility that a property holds no matter how many (finite) number of times the loop is executed. The property is frequently called the *loop invariant*, abbreviated INV. The rule of induction for loops in SDVS is:

Basis

[pre: **true**
env: (E)
mod:
post: ((#X = initial) & INV)]

Induction Step

[pre: ((initial $\leq$ .X $\leq$ final) & INV)
env: (E_1)
mod: (M_1)
post: ((#X = .X + 1) & INV(.\#))], E_1 disjoint from M_1

Conclusion

[pre: (initial $\leq$ final)
env: ($E \cup E_1$)
mod: M_1
post: ((#X = final) & INV (.\#))]

To demonstrate the induction rule consider the following ISPS program fragment called LOOP-DEMO:

```
B=1 NEXT                                                  1
C=A NEXT                                                  2
L := REPEAT                                               3
       BEGIN                                              4
       IF B > 100 => (LEAVE L) NEXT                       5
       B = B+1 NEXT                                       6
       C = C+A                                            7
       END                                                8
```

We want to prove if 1 $\leq$ B $\leq$ 100 then the loop invariant of LOOP-DEMO is (#C = #B*.A). B is the loop variable. If we can prove:

Basis

[pre: .upc = 1
env:
mod: upc
post: ((#B = 1) & (#C = #B*.A) & (#upc = 4))]

and

Induction Step

```
[pre: ((1 ≤ .B ≤ 100) & (.C = .B* .A) & (.upc = 4))
env:
mod: upc, B, C
post: ((#B=.B + 1) & (#C= #B* .A) & (#upc = 4))]
```

then we can conclude

```
[pre: .upc = 4
env:
mod:  upc, B, C
post: ((#B=100) & (#C = #B* . A) & (#upc = 4))]
```

thereby avoiding executing the loop 100 times.

The basis is true at the beginning of the repeat loop because

1) #B=1 as the result of the assignment statement on line 1

2) #C = #B* .A because #C = .A from line 2 and #C = .A = 1* .A = #B * .A

The induction step is true because

1) #B = .B + 1 from line 6

2) #C= #B*.A because #C=(.B*.A) + A from line 7 and #C=(.B*.A) + (.A*1) = (.B+1) *.A = #B*.A

4. Mapping

States in implementation the are a function of states in the specification. A mapping construct is necessary when state components in the specification have different names and structure than the corresponding state components in the implementation. This situation arises quite naturally in the microcode verification problem where the computer language specification and machine descriptions are derived independently. A mapping shows how the computer language behavior is implemented.

For example, consider using the program REGSWITCH as a specification. The theorem stating the correctness criterion of REGSWITCH can be written

```
[pre: (ISPS REGSWITCH)
      .REGSWITCH\upc=REGSWITCH\STARTED
env:
mod: ALL
post: #Y=.X #X=.Y
      #REGSWITCH\upc=REGSWITCH\HALTED]
```

where (ISPS REGSWITCH) is an abbreviation for the formula in Figure 6 (i.e., (ISPS REGSWITCH) abbreviates the set of state deltas resulting from the translation of the ISPS program), and STARTED and HALTED are SDVS defined labels for the first and last lines of the ISPS program.

Suppose REGSWITCH (an ISPS program) was implemented by the following ISPS program called IMPLSWITCH:

```
     IREGSWITCH(IX<15:0>,IY<15:0>,AC<15:0>) :=
        BEGIN

           ** DECLARATION.SECTION **
           ITEMP<15:0>

           ** SWAPPING.FUNCTION **
           ENTRY {MAIN} :=
              BEGIN
                 ITEMP = IY NEXT
                 AC = IY NEXT
                 IY = IX NEXT
                 IX = ITEMP
              END
        END
```

The variables X and Y are implemented as IX and IY, respectively. Each state change in REGSWITCH does not correspond to each state change in IMPLSWITCH, but the desired property that #Y = .X & #X = .Y holds at the end of IMPLSWITCH. The following theorem states that IMPLSWITCH implements the desired property that holds in REGSWITCH:

```
[pre: (ISPS IMPLSWITCH)
    (MAP X (IX) EQ)
    (MAP Y (IY) EQ)
    (MAP TEMP (IREGSWITCH\OTHERSTUFF))
    (MAP REGSWITCH\OTHERSTUFF (AC ITEMP))
    (MAP REGSWITCH\upc (IREGSWITCH\upc))
    (ONE-TO-ONE MAPFUNCTION)
    REGSWITCH\STARTED=MAPFUNCTION (IREGSWITCH\STARTED)
    REGSWITCH\HALTED=MAPFUNCTION(IREGSWITCH\HALTED)
env:
mod:
post: (ISPS REGSWITCH)]
```

where the "MAP" terms in the precondition of the theorem specify which objects in IMPLSWITCH implement objects in REGSWITCH, and empty environment and modification lists reduce the state delta to a formula that is an implication (i.e., pre ⇒ post). OTHERSTUFF is an SDVS defined variable; it does not appear in the ISPS programs. The variable TEMP in REGSWITCH is not implemented (i.e., it is an auxiliary variable). This is specified by mapping TEMP to OTHERSTUFF in IMPLSWITCH. Similarly, AC and ITEMP in IMPLSWITCH do not correspond to any variables in the specification, REGSWITCH. The predicate symbol EQ in a MAP construct means that the value of the variables in the construct are equal. Finally, ONE-TO-ONE specifies that its argument is a one-to-one function. Using the map construct, it is possible to prove that (1) one ISPS program is implemented by another program or (2) one ISPS program implements an abstraction of another program.

D. WHY MICROCODE VERIFICATION?

SDVS developers are focussing on microcode verification because microcode is used extensively in computer systems, and microcode is among the most heavily used software in computer installations. Additionally, microprogrammable hardware is becoming more complex and will have larger control stores. As a result, the design time, cost, and risk of error are increasing, and the necessity for reliable microcode is becoming more acute. Therefore, the development and application of analytical software verification techniques to microcode will have great payoff.

Prior to the advent of microcode, the hardware directly executed the computer instruction set. The architecture of the computer was established when the components of the computer were assembled. Thus, the computer's functional operation could not be changed except by hardware modification.

Microcode allows the design of a computer to be more flexible. Instead of directly executing the computer instruction set the hardware interprets the computer instruction set by executing microcode. In other words, microprogramming involves programming the control unit of a computer.

With microcode, there can be more than one instruction set on a single computer. This means that (1) programs written for older computers can run on new computers, (2) a computer can be tailored to a particular application (e.g., signal processing), and (3) new computer languages (e.g., 1750A) can be implemented on existing computers. Microcode also permits more than one computer model for an instruction set. Therefore, families of computers can have the same instruction set, allowing growth potential without reprogramming for each new computer installation. IBM took advantage of this concept when it introduced System 370; the System 370 has the same instruction set as the previous computer system, the System 360.

Not only is it important to focus on microcode verification because of the widespread use of microcode and the critical role microcode plays in computer systems, but application of verification methods to microcode reveals new issues in specification languages, in the relationship of specifications to implementations, and in the development of theories. A theory of bitstrings based on the theory of integers has been constructed for this application.

E. MICROCODE VERIFICATION

The diagram in Figure 8 represents the hierarchy of conceptual levels of a digital system that SDVS currently considers.

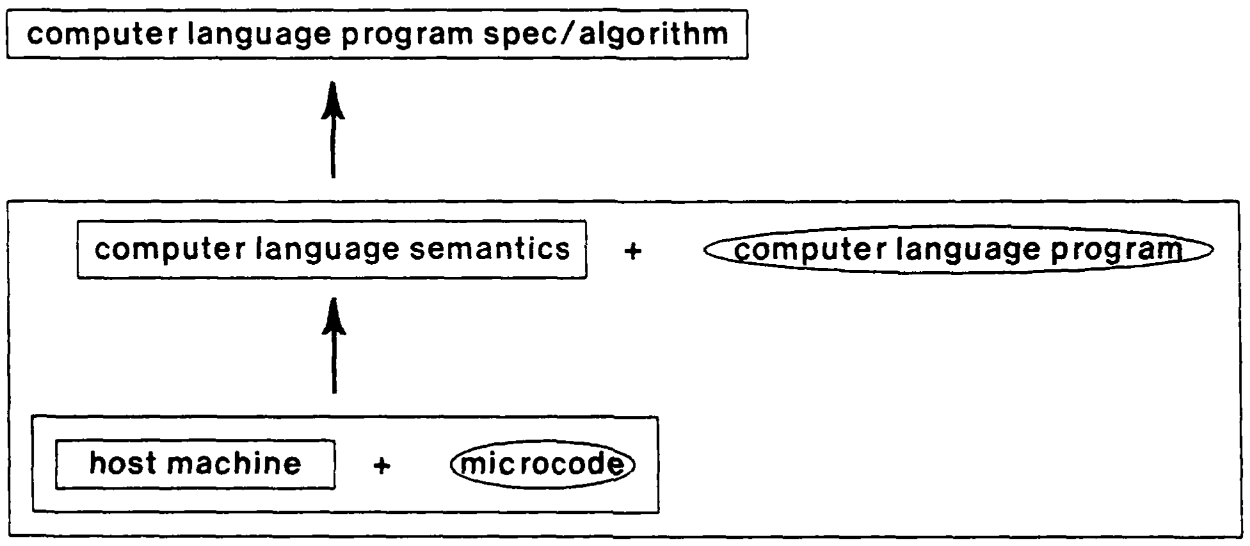

computer language program spec/algorithm
computer language semantics
+
computer language program
host machine
+
microcode

Figure 8: Hierarchy of Digital Systems

Each level of the hierarchy utilizes or is implemented by the level below it. The upward pointing arrows represent implementations; particular computer language semantics (behavior) are implemented by some host machine executing microcode and a program specification or algorithm is implemented by a computer language program that behaves in a predictable manner. The correctness criterion of each implementation is stated as a mathematical theorem.

Microcode verification is concerned with the relationship of the "computer language level" to the "microcode level". We want to show that a computer operating on some particular microcode correctly implements a computer instruction set.

The data structures at the microcode level include registers, control circuits, and data buses. The operations include transfers of data between data structures and arithmetic operations. Data structures and operations at the instruction set level are generally more abstract than at the microprogramming level. For example, at the instruction set level, the arithmetic\logical component of a computer can be viewed as a "black box": it performs arithmetic and logic operations when operands are supplied. At the microcode level, the arithmetic\logical unit consists of many registers and data paths manipulated by control circuits.

A typical instruction in a computer instruction set is a LOAD instruction. The execution of a LOAD instruction causes data in memory to be moved to a register. At the microcode level, the hardware decodes the LOAD instruction and executes a microprogram that will establish data paths among a network of registers and data buses to perform the loading function.

To prove that microcode correctly implements a computer instruction set we need to formally specify the behavior of the microcode in addition to specifying the behavior of the computer instructions. The specifications of the microcode behavior and the instruction set behavior are merely descriptions of the computer at different levels of abstraction. The behavior of the computer instructions is specified by defining the computer at a level of abstraction seen by the programmer. The specification of the computer at the microcode level is more detailed; it contains registers, operations, and programs hidden from the programmer. Our goal is to prove that the descriptions of the computer at different levels of abstraction are consistent.

The specifications of the computer can be described in the formal language of state deltas. Theorems can be deduced from these specifications. Alternatively, the specifications can be described in ISPS and then translated to state deltas; any computation that can be described in ISPS* can be described in terms of state deltas. The advantages of using ISPS are:

1) it is easier to write a specification in ISPS

2) a fast evaluation of the specifications can be made by using existing ISPS simulators and compilers

3) there exist many ISPS descriptions of computers

The theorem stating the microcode correctness criteria takes the form that the microcode behavior implies the instruction set behavior (i.e., the properties of the instruction set are preserved in its implementation, the microcode executing on the host machine). Both the host machine and the instruction set behavior are specified as ISPS programs. Therefore, the correctness proof involves showing that one program is correctly implemented by another program. Intuitively, it might be expected that all states and state changes at the instruction set level (the "target machine") must correspond to selected states and state changes at the microcode level (the "host machine"). However, only selected states and state changes in the target machine may correspond to those of the host machine because

1) the effect of a sequence of state changes may be implemented by a differently ordered sequence

2) auxiliary variables used to describe the target machine may not be implemented

3) various behaviors at the target level may not be describable in ISPS

---

* i.e., the subset of ISPS used in SDVS

It is expected that the third item above will occur in very few situations and those behaviors can be described in the logical language. Annotations (e.g., labels, logical formulas, lists of variables) inserted in the ISPS description of the target machine enable SDVS to extract the selected states and state changes. Thus, we anticipate that an annotated ISPS program of the target machine will suffice and SDVS will derive an "abstract specification" of the computer language semantics from the annotated ISPS program of the target machine.

Recall that a state consists of the current values of the variables, the value of the program counter, and the path condition. If the ISPS description of the machine is labeled, a state change results when the computation progresses from one label to the next. The labels in the program mark a path through the program, and thus implicitly denote the program counter and path condition. Consequently, the mapping from a state in one machine to a state in another machine is partitioned into two types of mapping: (1) labels in one ISPS description are mapped to labels in the other machine's ISPS description, and (2) variables of one ISPS description are mapped to variables of the other machine's ISPS description.

Figure 9 shows how SDVS is used to prove that the host machine executing particular microcode correctly implements the target machine.

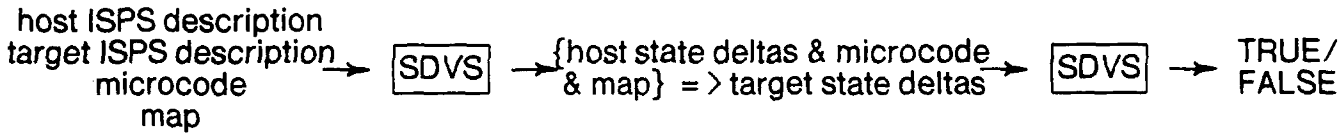


Figure 9: Microcode Verification Using SDVS

First, ISPS descriptions of the host and target, and a map of the target to the host are input into SDVS. The map includes a map of labels in the target to labels in the host (known in this paper as "map labels"). SDVS then generates state delta descriptions of the target and host from the ISPS descriptions. Conceptually, one state delta is initially created for each ISPS instruction. The state deltas of all instructions from one map label to the next map label in each description are then symbolically executed to form another set of state deltas. SDVS then constructs a new state delta of the form {host state deltas & microcode & map ⇒ target state deltas} where "host state deltas" refers to the state deltas describing the computation between map labels in the host machine (similarly for target state deltas). SDVS then assists the user in determining whether this state delta is true. If the state delta is true, we have proved that the microcode correctly implements the instruction set.

As mentioned above, the ISPS description of the target machine may be annotated because only selected states in the target description may correspond to states of the host. SDVS abstracts a specification from the annotated ISPS description. Call it the *abstract specification*. The abstract specification is consistent with the target machine and must be proved consistent with the host machine executing particular microcode. A typical abstract specification for the target ISPS description would be a state delta specification of each target instruction. Each state delta would be of the form

```
[pre: pre-instr
env:
mod: {target machine variables}
post: post-instr]
```

where pre-instr is a formula specifying the variables before execution of the target instruction, instr, and post-instr is a formula specifying the variables after execution of instr. The abstract specification can also be hand-generated.

In practice verification and simulation techniques are used in conjunction to achieve a high level of confidence that the microcode correctly implements the computer instruction set. The descriptions of the target and host machines are assumed to be correct in the verification method. These machine descriptions are large, and thus, it may be difficult to have a high level of confidence in them. Either simulation or a correctness proof with an abstract specification can increase the level of confidence in each machine description.

By supplying sample input values, simulation gives us a "quick and dirty" evaluation of each description. A certain degree of confidence in each specification is obtained with simulation, but there is usually a strong possibility of undetected errors. To increase the degree of confidence in each specification, each specification can be shown to be consistent with an abstract specification via symbolic execution and simplification. The abstract specification, defined as a state delta, may initialize input variables to data values and thus, achieve what simulation does. The abstract specification may not initialize any variables and the proof that each machine specification is consistent with the abstract specification may suffice for verification. And of course, the abstract specification may initialize some variables and not others, and thus, give us something less than complete verification, but something more than simulation. With confidence in each machine description, symbolic execution, simplification, and mapping are used to achieve a high level of confidence that the microcode correctly implements the instruction set.

SECTION II
A CASE STUDY

A. MACHINE DESCRIPTIONS

To discover and solve both theoretical and practical problems with microcode verification using SDVS, a small "toy" computer has been designed which contains many features relevant to a real computer. This computer, called the H-machine, is structured around a simplified version of the AM2901, a widely used microprogrammable arithmetic/logic unit (ALU).

In the manner previously discussed, the H-machine is specified at two levels of abstraction. The instruction set level description is called the target machine and the microcode level description is called the host machine.

The target machine is the computer architecture as seen by a programmer. Figure 10 contains a diagram of the target machine. It has a memory, TMEM, that stores both data and instructions, and four registers. Two of the registers, R1 and R2, can be referenced in target instructions; they can be accessed by the programmer. The other two registers, PC (program counter) and TIR (instruction register), are used for fetching, decoding, and executing target instructions. There are eight target instructions in the instruction set. Figure 11 lists the instructions and their descriptions.

Consider the LOAD R1 instruction. The execution of LOAD R1 causes data in TMEM to be moved to R1. Suppose there is "LOAD R1 15" at locations 0 and 1 in TMEM and PC contains the value 0. The first word of the instruction is fetched from TMEM and stored in TIR. PC is incremented to 1. The instruction is decoded. The second word of the instruction is fetched and stored in TIR. It contains the address, 15, where the data is located. PC is incremented to 2. The address is used to fetch the data and then the data is stored in R1. PC contains the location of the next instruction, and the cycle begins again.

Appendix A contains the ISPS description of the target machine. The main routine is a cycle of fetching, decoding, and executing target instructions.

Figure 12 depicts the same computer, but at the microcode level. In the host machine, machine components were added for fetching and decoding the microcode. Also, registers and data paths controlled by the microcode were added. Microcode programs in the microcode memory, UM, interpret target instructions. The format of the microinstruction is described in Figure 13. The ISPS description of the host machine is in Appendix B. The microcode initialization is in Appendix C.

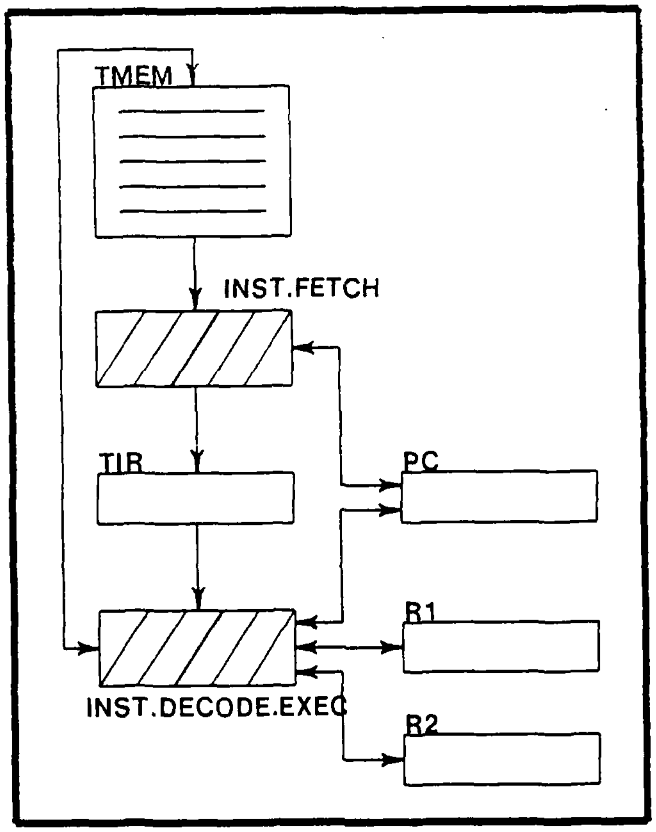


Figure 10: Target Machine

As an example, the LOAD R1 instruction is interpreted by executing 8 microinstructions, 3 of which are for fetching and decoding. The following summarizes the actions of the 8 microinstructions executed to perform the loading function:

1) MAR ← RAM.2901[15]; SEQ.MUX ← 1 *instruction fetch*

2) RAM.2901[15] ← RAM.2901[15] +1; HIR ← HMEM[MAR]; SEQ.MUX ← 2

3) SEQ.MUX ← INST.OP *decode*

4) MAR ← RAM.2901[15]; SEQ.MUX ← INST.OP + 1 *execution of LOAD R1*

5) RAM.2901[15] ← RAM.2901[15] + 1; MEM.IO ← HMEM [MAR]; SEQ.MUX ← INST.OP + 2

6) MAR ← MEM.IO; SEQ.MUX ← INST.OP + 3

7) MEM.IO ← HMEM[MAR]; SEQ.MUX ← INST.OP + 4

8) RAM.2901[1] ← MEM.IO; SEQ.MUX ← 0

Target Instruction	Abbreviation	Description
1000 | 0000 | 00000000 address	LOAD R1 address	R1 <- TMEM[address]
1001 | 0000 | 00000000 address	LOAD R2 address	R2 <- TMEM[address]
1010 | 0000 | 00000000 address	STORE R1 address	TMEM[address] <- R1
1011 | 0000 | 00000000 address	STORE R2 address	TMEM[address] <- R2
1100 | 0000 | 00000000	ADD	R1 <- R1 + R2
1101 | 0000 | 00000000	MULTBY5	R1 <- R1 * 5
1110 | 0000 | 00000000 address	JUMP address	PC <- TMEM[address]
1111 | 0000 | 00000000 address	STORE PC address	TMEM[address] <- PC + 1

Figure 11: Target Instructions

B. CORRECTNESS PROOF

As described in Section IE a mapping must be defined from the target machine to the host machine. To define the mapping, the main routine of the ISPS description of the target machine is labeled as follows:

```
                REPEAT
                      BEGIN
TLOOP:                      INST.FETCH() NEXT
                            INST.DECODE.EXEC()
                      END
```

The main routine of the host machine is labeled as follows:

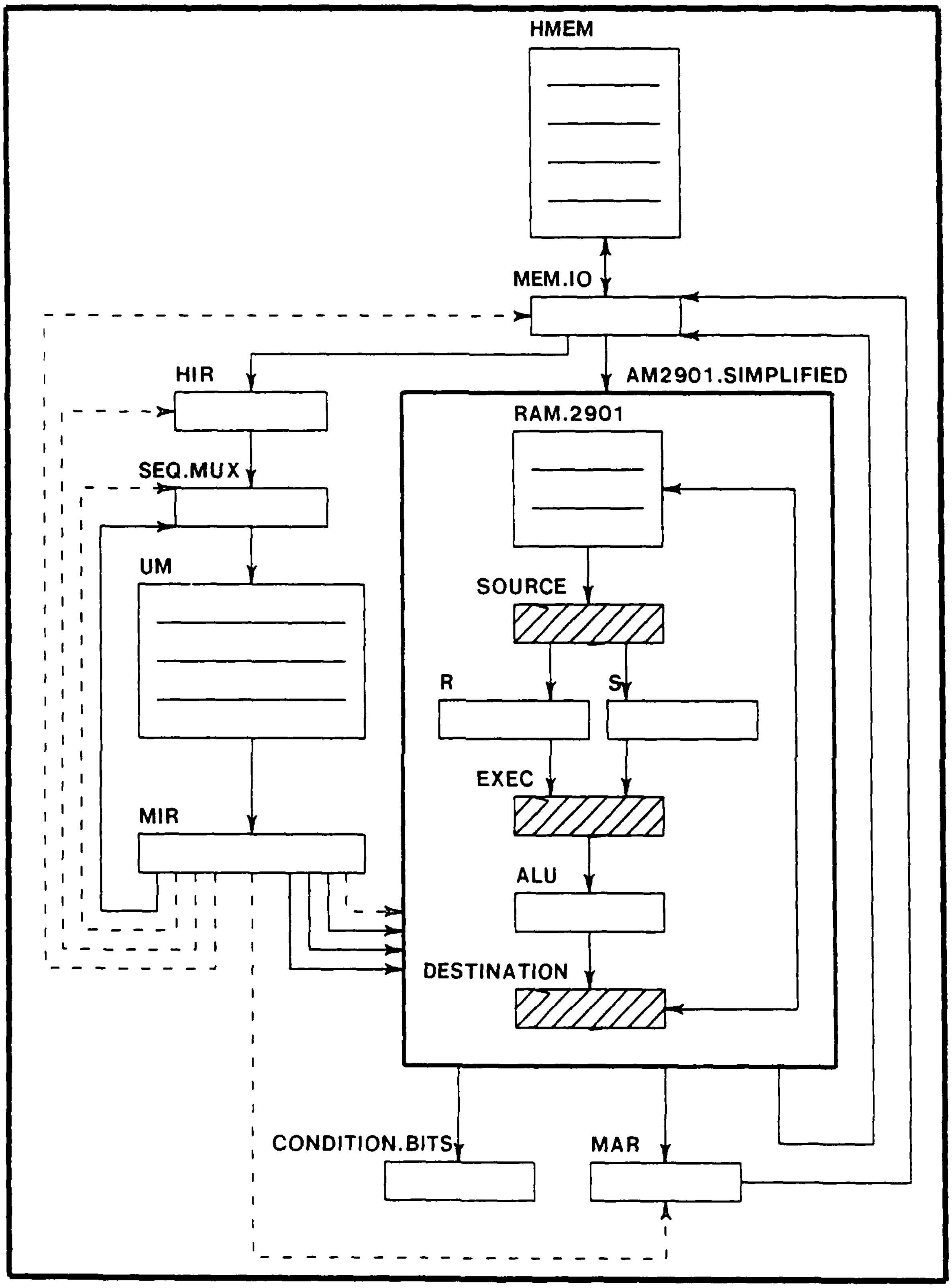


Figure 12: Host Machine

Bits	Field Name	Description
30-32	src.2901	ALU source(s)
27-29	op.2901	ALU operation
24-26	dest.2901	ALU destination(s)
23		spare
22	c.in	ALU carry-in
20-21	seq.mux.slct	control line for register SEQ.MUX (instruction sequence)
19	ld.ir	control line for register HIR
18	ld.mar	control line for register MAR
17	oe.db	control line for storing data in HMEM
16	r.wf	control line for fetching data from HMEM
12-15	A.adr	one of 16 registers in ALU
8-11	B.adr	one of 16 registers in ALU
0-7	next.adr	next microinstruction address

Figure 13: Microinstruction Format

```
                REPEAT
                     BEGIN
                          AMD2901.SIMPLIFIED(...) NEXT
                          MEM.IO(...) NEXT
                          SEQ.MUX(...) NEXT
                          MAR(...) NEXT
                          HIR(...) NEXT
      HLOOP:              MIR=UM[SEQ.MUX]
                     END
```

The state at TLOOP corresponds to the state at HLOOP where SEQ.MUX = 0, R1 is mapped to RAM.2901[1], R2 is mapped to RAM.2901[2], PC is mapped to RAM.2901[15], TIR is mapped to HIR, and TMEM is mapped to HMEM. This means the state in the target machine before a target instruction fetch must correspond to the state in the host machine before a target instruction fetch.

Symbolic execution of the target machine from TLOOP to TLOOP results in 8 state deltas, one for each target instruction. These state deltas comprise the abstract specification SDVS derives from the annotated ISPS description of the target machine. Each of these state deltas specifies the state change resulting from the execution of a target instruction on the target machine. Call these state deltas *target state deltas*. Between 5 and 8 ISPS instructions are symbolically executed to derive each target state delta.

Symbolic execution of the host machine from HLOOP to HLOOP specifies the state change resulting from the execution of a microinstruction. About 30-40 ISPS instructions are executed for each microinstruction and 4-8 microinstructions implement a target instruction. Therefore, about 100-300 ISPS instructions are executed for each target instruction in the host description compared with 5-8 ISPS instructions in the target description. The theorem stating the correctness criterion is:

Theorem 1:

```
[pre: (ISPS host)
    (MAP ....)
    (MICROCODE ....)
env:
mod:
post: (ISPS target)]
```

(ISPS host) is an abbreviation for the set of set deltas that result from the symbolic execution of the ISPS description of the host machine between map labels. The notation (ISPS host) is input to SDVS where "host" is the name of the file containing the ISPS description of the host machine. Similarly, for (ISPS target). (ISPS target) represents the SDVS derived abstract specification. (MAP ...) and (MICROCODE ...) abbreviate the mapping between machines and the microcode initialization, respectively. The current form of the map assertion for the H-machine is

```
(MAP PC (RAM.2901[15]) EQ)
(MAP R1 (RAM.2901[1]) EQ)
(MAP R2 (RAM.2901[2]) EQ)
(MAP TIR (HIR) EQ)
(MAP TMEM (HMEM) EQ)
(MAP TARGET
   (MACHINE\upc SEQ.MUX MACHINE\ROM UM RAM.2901[15] RAM.2901[1]
    RAM.2901[2] HMEM HIR))
(MAP TARGET\OTHERSTUFF
   (ENTRY IO.R.WF DATA.ENABLE MEMDATA ADR MEM.IO EXEC
    DESTINATION SOURCE P2901 RAM.2901[0] RAM.2901[3]
    RAM.2901[4:14] S R ALU C.IN.2901 I ALU.D.BUS B.ADR.2901
    A.ADR.2901 AMD2901.SIMPLIFIED CONDITION.BITS SELECT Y X MIR
    MARLOAD MARDATA MAR IRLOAD INSTRUCTION
    MACHINE\OTHERSTUFF))
(MAP TARGET\ROM (MACHINE\ROM UM))
(MAP TARGET\upc (MACHINE\upc SEQ.MUX)
```

Also, consider a hand-generated abstract specification for the target machine. The abstract specification for each target instruction on the target machine takes the form:

```
[pre: pre-instr
env:
mod: {target variables}
post: post-instr]
```

Appendix D contains a hand-generated abstract specification for the target machine of the H-machine example. Each of the eight state deltas in the abstract specification is given one of the following names: loadr1.sd, loadr2.sd, storer1.sd, storer2.sd, r1addr2.sd, r1times5.sd, jump.sd, and storepc.sd. The theorem asserting that the hand generated formulas are consistent with the ISPS description of the target machine is:

Theorem 2:

```
[pre:  ((ISPS TARGET.ISP))
env:
mod:
post: ((EVAL loadr1.sd) (EVAL loadr2.sd)
       (EVAL storer1.sd) (EVAL storer2.sd)
       (EVAL r1addr2.sd) (EVAL r1times5.sd)
       (EVAL jump.sd) (EVAL storepc.sd))]
```

The proof commands submitted to SDVS to prove Theorem 2 are

```
((readaxioms BSAXIOMS)
 (prove macro.sd
  (prove loadr1.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove loadr2.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove storer1.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove storer2.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove r1addr2.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove r1times5.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove jump.sd (rewrite |(.PC++1(16))<15:0>| chop) *)
  (prove storepc.sd (rewrite |(.PC++1(16))<15:0>| chop) *)))
```

The proof command "*" means symbolically execute the ISPS program until the goal is reached or execution has halted, and apply some axioms automatically. The "chop" axiom is not applied automatically and must be invoked by the user. The chop axiom is used to simplify bitstring expressions. In this example the application of the chop axiom allows SDVS to simplify the expression (.PC ++ 1 (16)) <15:0> to (.PC ++ 1(16)) because SDVS determines that adding the constant bitstring with value 1 and length 16 to the bitstring PC of length 16 will not overflow PC. The proof commands can be submitted interactively or processed as a batch job.

The same hand-generated state deltas can be proved consistent with the ISPS description of the host machine. Because the host machine uses names that are different from the abstract specification, a mapping must be specified.

SECTION III
FUTURE CONCERNS

Two concerns are mentioned for future consideration. The first concern is the feasibility of SDVS for real computers. Currently, it takes 1-2 hours to execute the LOAD R1 instruction on the host machine. The host machine is only 3 1/2 pages of ISPS code. A real machine can be more than 30 pages of ISPS code.

As in most verification systems, the majority of the time is spent in the theorem prover. In order to reduce the verification time, it appears that significant experimentation is necessary (1) to find a proper balance between automated deduction and user supplied proofs, (2) to construct a good set of axioms for the four theories, and (3) to develop heuristics (or tactics) for automated deduction. Also, this is a prototype system. It is anticipated that one or more orders of magnitude improvement can be made with appropriate software engineering.

The second concern is whether the target machine should be specified in ISPS. If target instructions can be easily specified with state deltas it may be desirable to eliminate the ISPS description of the target machine because

1) the ease of labeling ISPS descriptions and mapping labels depends on the style of the ISPS programmer, and

2) it is more feasible to symbolically execute and simplify one ISPS description than two ISPS descriptions

We are currently working on a way to abstract the computer language semantics from the ISPS description of the target machine in an efficient manner. SDVS will now be applied to larger examples, "real" computers. With this experimental data, performance issues will be addressed.

REFERENCES

[1] Mario R. Barbacci, Gary E. Barnes, Roderic G. Cattell, and Daniel P. Siewiorek.
The ISPS Computer Description Language.
CMU-CS-79-137, Carnegie-Mellon University, Computer Science Department, August, 1979.

[2] Stephen D. Crocker.
State Deltas: A Formalism for Representing Segments of Computation.
PhD thesis, University of California, Los Angeles, 1977.

[3] C. A. R. Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12(10):576-580, 583, October, 1969.

[4] Leo Marcus.
Dynamic and Static Reasoning in Program Verification.
Technical Report ATR-82(8478)-2, The Aerospace Corporation, June, 1982.

[5] L. Marcus and J. V. Cook.
SDVS User Manual.
Technical Report ATR-84(8478)-1, The Aerospace Corporation, 1984.

APPENDIX A
ISPS DESCRIPTION OF TARGET MACHINE

```
TARGET() :=
BEGIN
** GLOBAL.VARIABLES **
TIR<15:0>,
TMEM[0:127]<15:0>,
R1<15:0>,
R2<15:0>,
PC<15:0>
** MAPPINGS **
INST.OP<3:0> := TIR<15:12>
** MAIN.ROUTINE **
ENTRY() {MAIN} :=
    BEGIN
     REPEAT
      TLOOP:= BEGIN
        INST.FETCH() NEXT
        INST.DECODE.EXEC()
       END
    END
** FETCH.INSTRUCTION.FROM.MEMORY **
INST.FETCH() :=
    BEGIN
     TIR = TMEM[PC] NEXT
     PC = PC+1
    END
** DECODE.OPERATION.AND.EXECUTE **
INST.DECODE.EXEC() :=
    BEGIN
     DECODE INST.OP =>
       BEGIN
        '1000 := LOAD.R1 :=
           (INST.FETCH() NEXT R1 = TMEM[TIR]),
        '1001 := LOAD.R2 :=
           (INST.FETCH() NEXT R2 = TMEM[TIR]),
        '1010 := STORE.R1 :=
           (INST.FETCH() NEXT TMEM[TIR] = R1),
        '1011 := STORE.R2 :=
           (INST.FETCH() NEXT TMEM[TIR] = R2),
        '1100 := R1.ADD.R2 := (R1 = R1+R2),
        '1101 := R1.TIMES.5 := (R1 = R1*5),
        '1110 := JUMP :=
           (INST.FETCH() NEXT PC = TMEM[TIR]),
        '1111 := STORE.PC :=
           (INST.FETCH() NEXT TMEM[TIR] = PC)
       END
    END
```

APPENDIX B
ISPS DESCRIPTION OF HOST MACHINE

```
MACHINE() :=
 BEGIN
    ** REGISTERS**
    HIR(INSTRUCTION<15:0>.IRLOAD<>)<15:0> :=
        (IF IRLOAD => (HIR = INSTRUCTION)).
    MAR(MARDATA<15:0>.MARLOAD<>)<15:0> :=
        (IF MARLOAD => (MAR = MARDATA)).
    MIR<32:0>

    ** INSTRUCTION.FORMAT **
    INST.OP<7:0> := HIR<15:8>

    ** MICRO.INSTRUCTION.FORMAT  **
    I.ALU<8:0> := MIR<32:24>.
    OEYF<> := MIR<23>.
    C.IN<> := MIR<22>.
    SEQ.MUX.SLCT<1:0> := MIR<21:20>.
    LD.IR<> := MIR<19>.
    LD.MAR<> := MIR<18>.
    OE.DB<> := MIR<17>.
    R.WF<> := MIR<16>.
    A.ADR<3:0> := MIR<15:12>.
    B.ADR<3:0> := MIR<11:8>.
    NEXT.ADR<7:0> := MIR<7:0>

    ** MEMORIES **
    UM[0:255] <32:0>.
    HMEM[0:127] <15:0>

    ** MULTIPLEXERS **
    SEQ.MUX(X<8:0>.Y<8:0>.SELECT<1:0>)<8:0> :=
        BEGIN
            DECODE SELECT<0> =>
              BEGIN
                '0 := SEQ.MUX = X.
                '1 := SEQ.MUX = Y
              END
        END

     ** OUTPUT.OF.AMD2901.SIMPLIFIED **
     CONDITION.BITS<3:0>
     ** ALU.THIS.IS.A.SIMPLIFIED.AM2901 **
     AMD2901.SIMPLIFIED( A.ADR.2901<3:0>.B.ADR.2901<3:0>.
          ALU.D.BUS<15:0>. I<8:0>. C.IN.2901<1:0>)<15:0>: =
         BEGIN
         **ALU.OUTPUT**
```

```
FEQL0<>:= CONDITION.BITS<1>.
!OVR<>:=  CONDITION.BITS<2>.
SIGN<>:=  CONDITION.BITS<3>.
C.OUT<>:= CONDITION.BITS<0>

**ALU.INPUT**
 SRC.2901<2:0>:= I<8:6>.
 OP.2901<2:0>:=  I<5:3>.
 DEST.2901<2:0>:= I<2:0>

**ALU.LOCAL.VARIABLES**
  ALU<16:0>.
  R<15:0>.              ! R INPUT TO ALU
  S<15:0>               ! S INPUT TO ALU

**ALU.MEMORIES**
  RAM.2901[0:15]<15:0>

**MAPPING**
  F.2901<15:0>:=ALU<15:0>

**INSTRUCTION.CYCLE**
P2901() {MAIN}:=    !2901 PROCEDURE
  BEGIN
    SOURCE() NEXT
    EXEC() NEXT
    DESTINATION()
  END

**ACCESS.COMPUTATION**
 SOURCE() :=                ! SOURCE CALCULATION
   BEGIN
   DECODE SRC.2901 =>
     BEGIN
     #0 :=AQ:=  (R = RAM.2901[A.ADR.2901]  ).
     #1 :=AB:= (R = RAM.2901[A.ADR.2901] NEXT S = RAM.2901[B.ADR.2901]).
     #2 :=ZQ:= (R = 0 NEXT S=0     ).
     #3 :=ZB:=  (R = 0    NEXT S = RAM.2901[B.ADR.2901]).
     #4 :=ZA:=  (R = 0    NEXT S = RAM.2901[A.ADR.2901]).
     #5 :=DA:=  (R = ALU.D.BUS    NEXT S = RAM.2901[A.ADR.2901]).
     #6 :=DQ:= (R = ALU.D.BUS NEXT S=0      ).
     #7 :=DZ:=  (R = ALU.D.BUS    NEXT S = 0    )
     END
   END.

  DESTINATION() :=              ! DESTINATION CALCULATION
    BEGIN
    DECODE DEST.2901 =>
      BEGIN
      #0 :=DEST0.FY.FQ:=  ( AMD2901.SIMPLIFIED = F.2901 ).
```

```
            #1 := DEST1.FY := (AMD2901.SIMPLIFIED = F.2901 ).
            #2 := DEST2.AY.FB: = (AMD2901.SIMPLIFIED = RAM.2901[A.ADR.2901] NEXT
              RAM.2901[B.ADR.2901] = F.2901 ).
            #3 := DEST3.FY.FB: = (AMD2901.SIMPLIFIED = F.2901 NEXT
              RAM.2901[B.ADR.2901] = F.2901 ).
            #4 := DEST4.FY.SRBQ: = (AMD2901.SIMPLIFIED = F.2901 ).
            #5 := DEST5.FY.SRB: = (AMD2901.SIMPLIFIED = F.2901 ).
            #6 := DEST6.FY.SLBQ: = (AMD2901.SIMPLIFIED = F.2901 ).
            #7 := DEST7.FY.SLB: = (AMD2901.SIMPLIFIED = F.2901 )
            END
          END

      **INSTRUCTION.EXECUTION**
        EXEC() :=
          BEGIN
          DECODE OP.2901 =>
            BEGIN
            #0 := R.ADD.S: = ( ALU   = R + S +  C.IN.2901 ).
            #1 := S.SUB.R: = ( ALU   = S - R +  C.IN.2901 ).
            #2 := R.SUB.S: = ( ALU   = R - S +  C.IN.2901 ).
            #3 := R.OR.S: = ( ALU   = R OR S ).
            #4 := R.AND.S: = ( ALU   = R AND S ).
            #5 := R.MASK.S: = ( ALU   = NOT R AND S ).
            #6 := R.EXOR.S: = ( ALU   = R XOR S ).
            #7 := R.EXNOR.S: = ( ALU   = R EQV S )
            END NEXT
          FEQL0 = F.2901 EQL "0000 NEXT
          SIGN = F.2901<15> NEXT
          C.OUT = ALU<16>

          END

      END    ! END OF AM2901 DESCRIPTION


** MEMORY.IO **
MEM.IO(ADR<15:0>,MEMDATA<15:0>,DATA.ENABLE<>,IO.R.WF<>)<15:0>
     := BEGIN
          DECODE DATA.ENABLE@IO.R.WF =>
            BEGIN
              '00 := NO.OP().
                  !NO DATA ON BUS, BUT WRITE
                  ! => ERROR.BUT USE THIS AS NOP
              '10 := (HMEM[ADR] =MEMDATA NEXT
                   MEM.IO = MEMDATA).
                  !DATA AND WRITE
              '01 := MEM.IO = HMEM[ADR].
                  !NO DATA, BUT READ
              '11 := STOP()
                  !DATA, BUT READ => ERROR
            END
        END

        ** MACHINE.DESCRIPTION **
        ENTRY() {MAIN} :=
        BEGIN
            MIR = UM[0] NEXT
            REPEAT
                BEGIN
                  AMD2901.SIMPLIFIED(A.ADR.B.ADR.MEM.IO.I.ALU.'0 C.IN) NEXT
                  MEM.IO(MAR.AMD2901.SIMPLIFIED.OE.DB.R.WF) NEXT
                  SEQ.MUX(NEXT.ADR.INST.OP.SEQ.MUX.SLCT) NEXT
                  MAR(AMD2901.SIMPLIFIED.LD.MAR) NEXT
                  HIR(MEM.IO.LD.IR) NEXT
        HLOOP:=         MIR = UM[SEQ.MUX]
                END
        END
    END
```

APPENDIX C
MICROCODE

Microcode in Hexidecimal	Comments
	** INST.FETCH **
UM[0]="10004F001	MAR <- RAM.2901[15]
UM[1]="0C3490F02	RAM.2901[15] <- RAM.2901[15]+1; HIR <-HMEM[MAR]
	** DECODE **
UM[2]="000100000	SEQ.MUX <- INST.OP
	** LOAD.R1 **
UM["80]="10004F081	MAR <- RAM.2901[15]
UM["81]="0C3410F82	RAM.2901[15] <- RAM.2901[15]+1; MEM.IO <- HMEM[MAR]
UM["82]="1D8040083	MAR <- MEM.IO
UM["83]="000010084	MEM.IO <- HMEM[MAR]
UM["84]="1DB000100	RAM.2901[1] <- MEM.IO
	** LOAD.R2 **
UM["90]="10004F091	MAR <- RAM.2901[15]
UM["91]="0C3410F92	RAM.2901[15] <- RAM.2901[15]+1; MEM.IO <- HMEM[MAR]
UM["92]="1D8040093	MAR <- MEM.IO
UM["93]="000010094	MEM.IO <- HMEM[MAR]
UM["94]="1DB000200	RAM.2901[2] <- MEM.IO
	** STORE.R1 **
UM["A0]="10004F0A1	MAR <- RAM.2901[15]
UM["A1]="0C3410FA2	RAM.2901[15] <- RAM.2901[15]+1; MEM.IO <- HMEM[MAR]
UM["A2]="1D80400A3	MAR <- MEM.IO
UM["A3]="119021000	HMEM[MAR] <- RAM.2901[1]
	** STORE.R2 **
UM["B0]="10004F0B1	MAR <- RAM.2901[15]
UM["B1]="0C3410FB2	RAM.2901[15] <- RAM.2901[15]+1; MEM.IO <- HMEM[MAR]
UM["B2]="1D80400B3	MAR <- MEM.IO
UM["B3]="119022000	HMEM[MAR] <- RAM.2901[2]
	** R1.ADD.R2 **
UM["C0]="043002100	RAM.2901[1] <- RAM.2901[1]+RAM.2901[2]
	** R1.TIMES.5 **
UM["D0]="11B0013D1	RAM.2901[3] <- RAM.2901[1]
UM["D1]="0430031D2	RAM.2901[1] <- RAM.2901[1]+RAM.2901[3]
UM["D2]="0430031D3	RAM.2901[1] <- RAM.2901[1]+RAM.2901[3]
UM["D3]="0430031D4	RAM.2901[1] <- RAM.2901[1]+RAM.2901[3]
UM["D4]="043003100	RAM.2901[1] <- RAM.2901[1]+RAM.2901[3]
	** JUMP **
UM["E0]="10004F0E1	MAR <- RAM.2901[15]
UM["E1]="0C3410FE2	RAM.2901[15] <- RAM.2901[15]+1; MEM.IO <- HMEM[MAR]
UM["E2]="1D80400E3	MAR <- MEM.IO

```
UM["E3]="0000100E4              MEM.IO <- HMEM[MAR]
UM["E4]="1DB000F00              RAM.2901[15] <- MEM.IO
                                ** STORE.PC **
UM["F0]="10004F0F1              MAR <- RAM.2901[15]
UM["F1]="0C3410FF2              RAM.2901[15] <- RAM.2901[15]+1;
                                MEM.IO <- HMEM[MAR]
UM["F2]="1D80400F3              MAR <- MEM.IO
UM["F3]="10142F000              HMEM[MAR] <- RAM.2901[15]+1
```

APPENDIX D
ABSTRACT SPECIFICATION OF TARGET MACHINE

loadr1.sd:

```
[SD pre: (.TMEM[|.PC|]=LOAD.R1←OP   |.PC| lt 127   .TARGET\upc=TLOOP
         LOAD.R1←OP=32768(16)   |.TMEM[|(.PC++1(16))<15:0>|]| le 127)
         (PCOVERING TMEM[0:127] TMEM[.PC])
    env: (TARGET\ROM)
    mod: (R1 PC TARGET\upc TIR TARGET\OTHERSTUFF)
    post: (#R1=.TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]   #PC=(.PC++2(16))<15:0>)
         #TARGET\upc=TLOOP]
```

loadr2.sd:

```
[SD pre: (.TMEM[|.PC|]=LOAD.R2←OP   |.PC| lt 127   .TARGET\upc=TLOOP
         LOAD.R2←OP=36864(16)   |.TMEM[|(.PC++1(16))<15:0>|]| le 127)
         (PCOVERING TMEM[0:127] TMEM[.PC])
    env: (TARGET\ROM)
    mod: (R2 PC TARGET\upc TIR TARGET\OTHERSTUFF)
    post: (#R2=.TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]   #PC=(.PC++2(16))<15:0>)
         #TARGET\upc=TLOOP]
```

storer1.sd:

```
[SD pre: (.TMEM[|.PC|]=STORE.R1←OP   |.PC| lt 127   .TARGET\upc=TLOOP
         STORE.R1←OP=40960(16)   |.TMEM[|(.PC++1(16))<15:0>|]| le 127)
         (PCOVERING TMEM[0:127] TMEM[.PC])
    env: (TARGET\ROM)
    mod: (PC TARGET\upc TIR TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]
         TARGET\OTHERSTUFF)
    post: (#TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]=.R1   #PC=(.PC++2(16))<15:0>)
         #TARGET\upc=TLOOP]
```

storer2.sd:

```
[SD pre: (.TMEM[|.PC|]=STORE.R2←OP   |.PC| lt 127   .TARGET\upc=TLOOP
         STORE.R2←OP=45056(16)   |.TMEM[|(.PC++1(16))<15:0>|]| le 127)
         (PCOVERING TMEM[0:127] TMEM[.PC])
    env: (TARGET\ROM)
    mod: (PC TARGET\upc TIR TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]
         TARGET\OTHERSTUFF)
    post: (#TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]=.R2   #PC=(.PC++2(16))<15:0>)
         #TARGET\upc=TLOOP]
```

r1addr2.sd:

```
[SD pre: (.TMEM[|.PC|]=R1.ADD.R2←OP   |.PC| lt 127   .TARGET\upc=TLOOP
        R1.ADD.R2←OP=49152(16))
        (PCOVERING TMEM[0:127] TMEM[.PC])
   env: (TARGET\ROM)
   mod: (PC TARGET\upc TIR R1 TARGET\OTHERSTUFF)
   post: (#R1=(.R1++.R2)<15:0>   #PC=(.PC++1(16))<15:0>)
        #TARGET\upc=TLOOP]
```

r1times5.sd:

```
[SD pre: (.TMEM[|.PC|]=R1.TIMES.5←OP   |.PC| lt 127   .TARGET\upc=TLOOP
        R1.TIMES.5←OP=53248(16))
        (PCOVERING TMEM[0:127] TMEM[.PC])
   env: (TARGET\ROM)
   mod: (PC TARGET\upc TIR R1 TARGET\OTHERSTUFF)
   post: (#R1=(.R1**5(4))<15:0>   #PC=(.PC++1(16))<15:0>)
        #TARGET\upc=TLOOP]
```

jump.sd:

```
[SD pre: (.TMEM[|.PC|]=JUMP←OP   |.PC| lt 127   .TARGET\upc=TLOOP
        JUMP←OP=57344(16)   |.TMEM[|(.PC++1(16))<15:0>|]| le 127)
        (PCOVERING TMEM[0:127] TMEM[.PC])
   env: (TARGET\ROM)
   mod: (PC TARGET\upc TIR TARGET\OTHERSTUFF)
   post: (#PC=.TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|])
        #TARGET\upc=TLOOP]
```

storepc.sd:

```
[SD pre: (.TMEM[|.PC|]=STORE.PC←OP   |.PC| lt 127   .TARGET\upc=TLOOP
        STORE.PC←OP=61440(16)   |.TMEM[|(.PC++1(16))<15:0>|]| le 127)
        (PCOVERING TMEM[0:127] TMEM[.PC])
   env: (TARGET\ROM)
   mod: (PC TARGET\upc TIR TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]
        TARGET\OTHERSTUFF)
   post: (#TMEM[|.TMEM[|(.PC++1(16))<15:0>|]|]=(.PC++2(16))<15:0>
        #PC=(.PC++2(16))<15:0>) #TARGET\upc=TLOOP]
```

END

2-81

DTIC